Контрольная 3

Задача "maybe"

- управляемое конструирование, разрушение, присваивание
- семантика {copy, move} * {constructible, assignable}
- специализация для `maybe<T*>`
- конструирование с произвольными аргументами - `emplace(...)`

Сложности

- Выравнивание буфера, в котором конструируется хранимый объект
- Избыточные требования к хранимому типу
- Последовательности действий, реализующие конструктор/присваивание/reset/emplace
- Время жизни

Как сделать буфер под хранимый объект?

```
T obj_;  // (1) - сразу неправильно

char buf_[sizeof(T)];  // (2) -- здесь чего-то не хватае

union U {  // (3) -- не скомпилируется, нужно доработать
    T obj_;  // выравнивание нам гарантировано
};
U buf_;
```

Сырой буфер проще всего сделать так:

```
alignas(T) char buf_[sizeof(T)];
std::aligned_storage_t<sizeof(T), alignof(T)> buf_;  //

T& value() { return reinterpret_cast<T&>(buf_); }
const T& value() { return reinterpret_cast<const T&>(buf
```

Допилить напильником union

```
union U {
    T obj_;
    // нужны явные конструктор и деструктор
    U() {}
    ~U() {}
} buf_;

T& value() { return buf_.obj_; }
```

Как хранить признак "объект создан"?

- эффективное решение - `bool has_value_;`
- лентяйское решение - `T* ptr_;` (которое всегда или nullptr, или &buf_)

Требования к типу объекта.

```
maybe<T> x;  // единственное требование - Destructible

maybe<T> y(T{});  // CopyConstructible / MoveConstructib
maybe<T> z(y);  // CopyConstructible
z = T{};  // ???
z = y;  // ???
z = std::move(y);  // ???
z.reset();
z.reset(T{});
z.emplace(A{}, B{});
```

Как возникают избыточные требования?

```
maybe(const maybe& other)
    // : maybe()
{
    *this = other;
}
```

Последовательность действий

Исходные условия для операции

- над одним объектом: {пусто, размещено}
- над приёмником и источником: {пусто, размещено} * {пусто, размещено}

Выполняемые действия

Конструкторы

- Приёмник исходно пуст. Можно не ветвиться и не проверять, это факт.
- Если источник - T, то он исходно существует :)
- Если источник - maybe<T>, то он может быть в любом состоянии.

`reset(T)` И `emplace(…)`

- Приёмник может быть в любом состоянии
- Источник существует

Присваивание - самая сложная операция.

- Приёмник пуст, источник пуст - отдыхаем
- Приёмник пуст, источник существует - конструируем
- Приёмник существует, источник пуст - разрушаем
- Приёмник существует, источник существует - ???

Неправильное решение: сводим задачу к предыдущей, разрушаем источник, а затем, если надо, конструируем.

Правильное решение: выполняем присваивание.

Типичные ошибки

- Забыли разрушить, сконструировали поверх
- Разрушили и присвоили поверх
- Флажок сняли/поставили, а разрушить/сконструировать забыли (или наоборот)

Семантика перемещения

Перемещение бывает двух видов: "слабая" и "сильная"

"Слабая" семантика: источник остаётся в каком-то валидном состоянии, лишь бы операция была быстрой и noexcept.

В частности, может подойти и copy, и swap.

Все тривиальные типы имеют слабую семантику перемещения.

`std::optional` также имеет слабую семантику. Прямо оговаривается, что деструктор у источника не вызывается.
https://en.cppreference.com/w/cpp/utility/optional/optional

"Сильная" семантика: источник переходит в пустое состояние.

Все стандартные умные указатели.

Стандартные контейнеры не обязаны иметь сильную семантику, но, как правило, имеют. (кроме std::array, - у него нет пустого состояния).

Несмотря на то, что maybe - это рукодельный `std::optional`, в техзадании не было ограничений на семантику. Можно и так, и этак.

Разумеется, перемещение хранимого типа должно быть честным.

Если T - это `std::unique_ptr`, например, то конструктор `maybe<T>(T&& other)` должен переместить аргумент, сделав `std::move(other)`.

Но конструктор `maybe<T>(maybe<T>&& other)` может, на выбор